Thoughts on Interactions in Distributed Services

Jack Haverty

MIT Laboratory For Computer Science

September 16, 1976

I. ABSTRACT

This paper addresses some issues concerned with the design of distributed services. In particular, it is concerned with the characteristics of the interactions between programs which support some service at various network sites. The ideas presented are derived mainly from experience with various service protocols [Reference 1] on the ARPANET.

A model is developed of interactions between programs. Salient features of this model which promote and simplify the construction of reliable, responsive services are identified. These qualities are motivated by problems experienced with various ARPANET protocols and in the design and maintenance of programs which use these protocols in the performance of some service.

Using this model as a template, the general architecture of one possible interaction protocol is presented. This mechanism provides a foundation on which protocols would be constructed for particular services, simplifying the process of creating services which are easy to implement and maintain, and appear reliable and responsive to the customer. This presentation is meant to serve as an introduction to a specific instance of such a protocol, called the RRP, which is defined in one of the references.

II. OVERVIEW AND TERMINOLOGY

This paper considers the interaction of two programs which support some network service. It develops a model of the interactions of a class of such applications, and includes some thoughts on desirable goals and characteristics of implementations. The model is derived from a proposal [Reference 2] for mail-handling systems. Terminology, as introduced, is highlighted by capitalization.

Many uses of computer networks involve communication directly between programs, without human intervention or monitoring. Some examples would include an advanced mail-handling system, or any kind of multi-site data base manager.

Such programs will be termed SERVERs. They are the users of some mechanism which provides the needed communication and synchronization. The particular facility which the servers implement will be termed a SERVICE. Servers for any particular service may be written in several languages, operate in various system environments on different kinds of computers. The entity which utilizes the service will be termed the CUSTOMER.

Servers interact during ENCOUNTERs, which are the periods when two servers are in communication. An encounter begins when one server establishes a CHANNEL, a bidirectional communication link with another server. The interaction between servers is effected by the exchange of information over the channel. The conventions used in such an exchange are defined by the PROTOCOLs for the interaction.

The theme of this paper is a model for a particular class of process interactions which may be used as a basis for many possible services, where the interactions are fairly simple. Services which fit in this category interact in a manner which can be modeled by a REQUEST-REPLY DISCIPLINE, which is defined herein.

A set of guidelines and goals is developed, which address issues relevant to ease of implementation and reliability of operation of servers. These guidelines may be used to assist in the formulation of protocols specific to appropriate services.

Additionally, the guidelines presented may be used as a basis for a general process interaction protocol, by extracting the primitives and operational concepts which would be common to a protocol constructed for virtually any such service.

From these ideas, a protocol which provides a foundation can be constructed, to be extended for particular services by adding primitives specific to each. The RRP [Reference 4] is one such possible protocol. It provides basic primitives to control the interaction between servers, and a mechanism for extending the primitives to include service-specific operations.

The discussion here is primarily intended to explain the basis for the design of the RRP, and to present some general issues of design of services.

III. THE REQUEST-REPLY DISCIPLINE

The class of services relevant to this discussion are those whose interactions could be performed in the following manner.

Two servers have established a channel by some external means. A single interaction between servers begins with one server, called the REQUESTER, issuing a request. The server receiving that request, the RESPONDER, issues a REPLY. The requester interprets the reply sequence to determine whether the request was successful, failed, or partially failed, and takes appropriate action. Such a sequence of events is termed an EXCHANGE. This is analogous to a subroutine call in a simple single-processor operating system.

This model is termed a REQUEST-REPLY DISCIPLINE of program interaction. It should be noted that this is only a model of program behavior, and does not necessarily exclude services which require, for example, some measure of pipelining of requests for efficiency in long-delay situations. In fact, most network services would require such measures, but their interactions can still be reduced to the request-reply model.

At any time, one of the partners is in control of the interaction, and is termed the MASTER of the interaction. The other partner is called the SLAVE. In the simplest cases, the requester is always the master, although this is not always true in particular implementations, such as the RRP [Reference 4].

IV. CHARACTERISTICS OF AN INTERACTION MECHANISM

The following set of characteristics desirable in an interaction mechanism is the result of experience with program communication in various ARPANET applications, such as message services, file transfer, Datacomputer, and remote job entry applications.

In attempting to produce such systems, several qualities recurred which would be desirable in the substructure upon which the systems are built. These characteristics would promote ease of writing and debugging servers, maintaining reliability, and providing services which are responsive to customer needs, while avoiding disruptions of service.

The qualities desired in the interaction mechanism are presented along with a discussion of the effects which they are intended to produce in the associated services. It must be emphasized that this discussion is related to a class of simple services, and might nct be appropriate for more complex applications.

1/ Servers must be able to transfer data in a precise fashion, retaining the structure and semantic meaning of the data, despite the dissimilarities of the computer systems in which they function.

2/ Synchronization and timing problems due to the characteristics of the communications link must be isolated and handled separately from any which might be characteristic of the service itself.

3/ Since services may wish to provide expanded facilities as they are used and developed, a mechanism must be included to enable the service protocol to evolve.

4/ Since various programs which act as servers may undergo simultaneous development, care must be taken to insure that servers with different capabilities interact reliably, maintaining at least the same level of service as existed previously.

5/ The mechanisms for extending the facilities must

avoid requiring servers to be modified when new capabilities are introduced, but not impede progress by maintainers who are anxious to provide a new or experimental service.

These qualities may be placed in three categories, data precision (1), process synchronization (2), and service enhancement (3, 4, 5). Each will be discussed separately in the following sections. The significance of each quality and its effect on the associated service characteristics will be included, with some references to related problems with current and past services.

Since these considerations are common to many possible services, it is appropriate for the interaction protocol to include them within its machinery as much as possible. This permits services to be implemented which, if carefully designed, inherit these properties from the interaction substrate.

V. PRECISE DATA TRANSFER

Precision in data transfer permits semantic and structural information which exists in the sender's instance of a datum to be reproduced in the receiver's image of the datum, even though it may be represented in the systems involved in entirely different fashions.

For programs to provide powerful, reliable capabilities, they must be able to interact using data which is meaningful to the particular service involved. The interaction mechanism must permit services to define their own relevant data types, and transfer such items efficiently and precisely. This facility provides a 'standard' for data, permitting the service's designers to concentrate on higher-level issues of concern to the service itself.

Data of a given type should be recognizable as such without need for context. The mechanism should also permit new data types to be handled by older servers without error, even though they cannot interpret the semantics of the new data.

These characteristic permits services to be designed in terms of the abstract data they need to function, without continued detailed concern for the particular formats in which it is represented within various machines.

For example, servers may need to transfer a datum identifying a particular date, which may be represented internally within systems in different forms. The data transfer mechanism should be capable of transferring such a datum as a date per se, rather than a strict pattern of bits or characters.

For example, in current FTP-based mail systems, messages often contain information with significant semantic meaning, which is lost or obscured when transferred between sites. An example might be a file specification, which effectively loses all identity as such when translated into a simple character stream. People can usually recognize such streams as file names, but it is often extremely difficult, time-consuming, and inefficient to construct a program to do so reliably. As a result, services which should be easy to provide to the customer, such as automatic retrieval of relevant files, become difficult and unreliable.

Some success has been achieved in handling certain data, such as dates and times, by defining a particular character pattern which, if seen in a particular context, can be recognized as a date or time. Each of these cases has been done on an individual basis, by defining a format for the individual data of concern. Generally, the format depends to some extent on the datum occurring within a particular context, and is not unique enough to be identifiable outside of that context.

A particular service can achieve data precision by meticulous specification of the protocols by which data is transferred. This need is widespread enough, however, that it is appropriate to consider inclusion of a facility to provide data precision within the interaction mechanism itself.

The major effect of this would be to facilitate the design of reliable, responsive services, by relieving the service's designers from the need to consider very low-level details of data representation, which are usually the least interesting, but highly critical, aspects of the design. By isolating the data transfer mechanism, this architecture also promotes modularity of implementations, which can reduce the cost and time needed to implement or modify services.

VI. PROCESS SYNCHRONIZATION

A major source of problems in many services involves synchronization of servers interacting over a relatively low-bandwidth, high-delay communications link.

Interactions in most services involve issuing a command and waiting for a response. The number of responses which can be elicited by a given command often varies, and there is usually no way to determine if all replies have arrived. Programs can easily issue a request before the responses to a previous request have completed, and get out of synchronization if a response is incorrectly matched to a request. Each server program must be meticulously designed to be capable of recovering if an unexpected reply arrives after a subsequent command is issued.

Note that, for reliable operation, it is always necessary that each response cause a reply to occur, at least in the sense that the request is confirmed at some point. No service should perform a critical operation, such as deleting a file, which depends on the success of a previous request unless it has been confirmed. Requests in current protocols which do not appear to cause a reply may be viewed as confirmed later when a subsequent request is acknowledged. While such protocols work, they are more opaque than desirable, and consequently more difficult to implement.

These characteristics of protocols have often resulted in implementation of ad hoc methods for interaction, such as timeouts of sufficient length to assure correctness in an acceptably high percentage of situations. Often this has required careful tuning of programs as experience in using a protocol shows which commands are most likely to cause problems. Such methods generally result in a service which is less responsive, powerful, or efficient than desirable, and expensive to build and maintain.

Additionally, protocol specifications for services have often been incomplete, in that an enumeration of the responses which may occur for a given command is inaccurate or non-existent. This greatly complicates the task of the programmer trying to construct an intelligent server. In most cases, servers are slowly improved over time as experience shows which responses are common in each instance.

The synchronization problems mentioned above are in addition to those which naturally occur as part of the service' operation. Thus, problems of synchronization may be split into two classes, those inherent in the service, and those associated with the interaction mechanism itself.

Construction of reliable, responsive servers can be assisted by careful design of the interaction mechanism and protocols. An unambiguous, completely specified mapping between commands and responses is desirable. Synchronization considerations of the two types can be attacked separately. An interaction mechanism which handles its own synchronization can be provided as a base for service' designers to use, relieving them of considerations of the low-level, protocol-derived problems, by providing primitives which encourage the design of reliable services.

To achieve a reasonable effective bandwidth, it is usually desirable to permit interacting programs to operate in a full-duplex fashion. Significant amounts of data are often in transit at any time. This magnifies the problems associated with interaction by introducing parallelism. The interaction mechanism can also be structured to provide ways of handling these problems, and to provide a basis on which servers which exploit parallelism can be constructed.

Many of these problems are too complex to warrant their consideration in any but the most active services. As a result, services are often constructed which avoid the problems by inefficiencies in their operation, as mentioned above. Provision of an interaction mechanism and primitives for use by such services would promote efficient interaction even by simple services which do not have the resources to consider all the problems in detail.

VII. SERVICE ENHANCEMENT

When particular programs implementing a service are undergoing development simultaneously by several organizations, or are maintained at many distributed sites, many problems can develop concerning the compatibility of dissimilar servers.

This situation occurs in the initial phase of implementing a service, as well as whenever the protocols are modified to fix problems or expand the service. Virtually every interaction protocol is modified from time to time to add new capabilities. Two particular examples are the TELNET protocol and mail header formats.

In most cases, the basic protocol had no facility for implementing changes in an invisible fashion. This has had several consequences.

First, it is very difficult to change a protocol unless the majority of concerned maintainers are interested in the changes and therefore willing to exert effort to change the programs involved. This situation has occurred in previous cases because any change necessarily impacts all servers. The services involved therefore often stagnate, and it becomes inappropriately difficult to provide a customer with a seemingly simple enhancement.

Second, when protocols change by will of the majority, existing servers often stop working or behave erratically when they suddenly find their partner speaking a new language. This is equally disconcerting to the service' customer, as well as annoying to the maintainers of the servers at the various sites affected.

These problems can be easily avoided, or at least significantly reduced, by careful design of the interaction protocols. In particular, the interaction mechanism itself can be structured to avoid the problem entirely, leaving only those problems associated with the particular service' operations themselves.

The interaction machinery should be structured so that the mechanisms of the interaction substrate itself may be improved or expanded in a fashion which is absolutely invisible to current servers.

1/ No server should be required to implement a change which is unimportant to its customers.

2/ No server should be prevented from utilizing a new facility when interacting with a willing partner.

3/ Service should not be degraded in any way when a new protocol facility is made available.

In cases where a single service is provided by different server programs at many sites, it is desirable for the various sites to be able to participate at a level appropriate to them. A new server program should be able to participate quickly, using only simple mechanisms of the protocol, and evolve into more advanced, powerful, or efficient interaction as desired. Sites wishing to utilize advanced or experimental features must be allowed to do so without imposing implementation of such features on others. Conversely, sites wishing to participate in a minimal fashion must not prevent others from using advanced features. In all cases, the various servers must be capable of continued interaction at the highest level supported by both.

The goal is an evolving system which maintains reliability as well as both upward and downward compatibility. The protocol itself should have these characteristics, and it should provide the mechanisms to permit service interaction protocols to be defined which inherit these qualities.

VIII. STRUCTURING AN INTERACTION MECHANISM

The qualities presented previously should provide at least a starting point for implementation of services which avoid the problems mentioned. The rest of this paper addresses issues of a particular possible architecture of an interaction mechanism.

The design architecture splits the service-specific conventions from those of the interaction per se. An interaction protocol is provided which implements the machinery of the request-reply model, and includes handling of the problems of data precision, synchronization, and enhancement. This protocol is not specific to any service, but rather provides primitives, in the form of service-designed requests and replies, on which a particular service protocol is built.

An actual implementation for a particular service could merge the code of the interaction protocol with the service itself, or the interaction protocol could be provided as a 'service' whose customer is the service being implemented.

The goals of this design architecture follow.

1/ Provision of a general interaction mechanism to be used by services which follow a request-reply discipline. Services would design their protocols using the primitives of the mechanism as a foundation.

2/ INTERACTION MECHANISM EXTENSIBILITY. The interaction mechanism may be expanded as desired without impacting on existing servers unless they wish to use the new features.

3/ SERVER EXTENSIBILITY. Servers can be implemented using the most basic primitives. Implementations may later be extended to utilize additional capabilities to negate some of the inefficiency inherent in a strict request-reply structure.

4/ SERVICE EXTENSIBILITY. The primitives permit a service to be expanded as desired without impacting on existing servers in any way unless they wish to use the new features.

5/ SERVER COMPATIBILITY. Within the set of servers of a given application, it is possible to have different servers operating at different levels of sophistication, and still maintain the ability for any pair of servers to interact successfully.

These goals involve two basic areas of design. First, the interaction mechanism itself is designed to meet the goals. Secondly, guidelines for structure of the particular service' protocols are necessary, in order for it to inherit the qualities needed to meet the goals.

IX. PARTITIONING THE PROBLEM

In defining the interaction mechanism itself, the problem may be simplified by considering two areas of concern separately.

1/ The characteristics and format of the data conveyed by the channel may be defined.

2/ The conventions used to define the interaction may be defined, in terms of the available data supported by the channel.

For purposes of this paper, the data repertoire and characteristics of the channel are assumed to be as described in [Reference 3] and summarized in an appendix. Discussions of the interaction between servers will use only the abstract concepts of primitive and semantic data items, to isolate the issues of interaction from those of data formats and communication details, and therefcre simplify the problem.

Additionally, actual implementation of a mechanism following the ideas presented here can be accomplished in a modular fashion, isolating code which is concerned with the channel itself from code concerned with the interaction behavicr.

The interaction mechanism provides primitives to the service' designer which are likewise defined in terms of the data items available. Service designers are encouraged, but not required, to define interactions in terms of these data only.

X. THE PRIMITIVES

The interaction mechanism assumes the existence of a channel [Reference 3] between two servers. Two new semantic data types are defined to implement the interaction. These are, unsurprisingly, called CONTROL REQUESTs and CONTROL REPLYs. Each of these data items contains at least two elements.

1/ The TYPE element identifies a particular type of request or reply.

2/ The SEQUENCE element is used to match replies to their corresponding request.

Other elements may appear. Their interpretation depends on the particular type of request or reply in which they appear.

The interaction protocol itself is defined in terms of control requests and control replies. A very small number of request and reply types is defined as the minimal implementation level. Additional request and reply types are also defined, for use by more advanced servers, to provide additional capabilities to the service, or simply to increase efficiency of operation.

Two additional data items are defined, called USER REQUESTs and USER REPLYs. These are structured like requests and replies, but the various types are defined by the service itself, to implement the primitives needed in its operation.

Control and user requests and replies are generically referenced as simply REQUESTs and REPLYs.

The protocol of the interaction has several characteristics which form the basis of the request-reply model, and attempt to meet the goals mentioned previously.

1/ Every request elicits a reply.

2/ Every reply is associated unambiguously with a previous request.

3/ Each server always knows the state of the interaction, such as whether or not more data is expected from its partner.

4/ The protocol definition includes enumeration of the possible responses for each request. Service protocols are encouraged to do likewise for user requests and user replies.

5/ Servers who receive requests of unknown type issue a response which effectively refuses the request. Servers attempting to use advanced features of a protocol 'rephrase' their requests in simpler terms where possible, to maintain the previous level of service.

The minimal implementation will support interaction almost exactly along the lines of the request-reply discipline.

Extensions to the minimal configuration are defined for two reasons. First, the strict request-reply discipline model is inefficient for use in high-volume situations because of the delays involved. Several extensions are defined to cope with this problem. Thus, although the interaction is based on such a discipline, it does not necessarily implement the interaction in that fashion. Second, additional primitives are defined which provide some standard process synchronization operations, for use by the services.

The protocol architecture presented here is defined in detail in an associated document. [Reference 4]

Appendix i -- The Channel

The following discussion is a summary of the ideas presented in [Reference 3], which should be consulted for further detail.

The communication link between two servers is termed a CHANNEL. Channels provide bidirecticnal communications capabilities, and will usually be full-duplex. The programs involved establish the channel as their individual applicaticns require, using some form of initial connection prctoccl.

The channel acts as an interface between servers. It conveys abstract data items whose semantics are understood by the programmers involved, such as INTEGERs, STRINGs, FILE PATH NAMEs, and so on. Because the users of the channel may operate in dissimilar computer environments, their communication is defined only in terms of such abstract data items, which are the atomic units of information carried on the channel. The program implementing the channel at each site converts the data between an encoded transmission format appropriate to the particular communication link involved, and whatever internal representational form is appropriate in the computer itself.

The channel protocol provides a mechanism for definition of various types of data items of semantic value for the particular service concerned, for example, possibly, user-name, set, syllable, sentence, and other data items of interest to the particular service. The channel provides a mechanism for transportation of such user-defined data from host tc host.

The channel may actually be implemented by one or more separate encoding mechanisms which would be used in different conditions. Initially, the channel machinery would provide a rudimentary facility based on a single mechanism such as the MSDTP [Reference 3].

The mechanism is not dependent on the existence of actual line-style network connections but will operate in other environments, such as a message-oriented (as opposed to connection-oriented) communications architecture, and in fact is more naturally structured for such an environment.

XI. REFERENCES

[1] Network Information Center, ARPANET Protocol Handbook, April, 1976.

[2] Broos, Haverty, Vezza, Message Services Protocol Proposal, December, 1975.

[3] Haverty, Jack, Message Services Data Transfer Protocol, NWG RFC 713, NIC 34729, April, 1976.

[4] Haverty, Jack, RRP, A Process Communication Protocol for Request-Reply Disciplines, NWG RFC 723, NIC 36807, (to be issued)